INFO6205 Spring 2022 Project

27-APR-2022

MENACE

TEAM MEMBERS

Arjun Raja Yogidas - 002964082

Koovehithilu Shrikrishna Joisa - 002920963

Introduction

Tic-tac-toe, noughts and crosses, or Xs and Os is a paper-and-pencil game for two players who take turns marking the spaces in a three-by-three grid with X or O. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row is the winner. It is a solved game, with a forced draw assuming best play from both players.

An early variation of tic-tac-toe was played in the Roman Empire, around the first century BC. It was called terni lapilli (three pebbles at a time) and instead of having any number of pieces, each player had only three; thus, they had to move them around to empty spaces to keep playing.The game's grid markings have been found chalked all over Rome. Another closely related ancient game is three men's morris which is also played on a simple grid and requires three pieces in a row to finish, and Picaria, a game of the Puebloans. More Reading about the history can be found here.

The MENACE

In 1961, Donald Michie built MENACE (Machine Educable Noughts And Crosses Engine), a machine capable of learning to be a better player of Noughts and Crosses. As computers were less widely available at the time, MENACE was built from 304 matchboxes.

Initially, MENACE begins with four beads of each color in the first move box, three in the third move boxes, two in the fifth move boxes and one in the final move boxes. Removing one bead from each box on losing means that later moves are more heavily discouraged. This helps MENACE learn more quickly, as the later moves are more likely to have led to the loss.

After a few games have been played, it is possible that some boxes may end up empty. If one of these boxes is to be used, then MENACE resigns. When playing against skilled players, it is possible that the first move box runs out of beads. In this case, MENACE should be reset with more beads in the earlier boxes to give it more time to learn before it starts resigning. More reading here

AIM

The aim of this project is to implement “The Menace” by replacing matchboxes with values in a hash table having the key as the different states of the game.

APPROACH

Using the board state after each move as the key in a hash table, we choose the next move. The next move is determined by training the system using a Reinforcement learning algorithm, that incentivizes the algorithm every-time it makes a correct move.

We have used Q-learning, a model free reinforcement algorithm to determine the rewards for each action performed by the algorithm. We chose to use this algorithm because we wanted to use an algorithm which does not use the transition probability distribution associated with the Markov decision process (MDP). A model-free RL algorithm can be thought of as an "explicit" trial-and-error algorithm, thus we used a trial and error approach to train the model.

Q-Learning Algorithm

Reinforcement learning involves an agent, a set of states **S**, and a set **A** of actions per state. By performing an action a ε **A**, the agent transitions from state to state. Executing an action in a specific state provides the agent with a reward (a numerical score).

After Δt steps into the future the agent will decide some next step. The weight for this step is calculated as ɣΔt, where ɣ (the discount factor) is a number between 0 and 1 (0 <= ɣ <= 1) and has the effect of valuing rewards received earlier higher than those received later (reflecting the value of a "good start"). ɣ may also be interpreted as the probability to succeed (or survive) at every step Δt.

The algorithm, therefore, has a function that calculates the quality of a state–action combination:

Q: **S x A** → **R**

Before learning begins, Q is initialized to a possibly arbitrary fixed value (chosen by the programmer). Then, at each time *t* the agent selects an action a_t observes a reward r_t enters a new state s_{t+1} (that may depend on both the previous state s_t and the selected action), and Q is updated. The core of the algorithm is a Bellman equation as a simple value iteration update, using the weighted average of the old value and the new information.

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right.}^{\text{temporal difference}} \underbrace{\left. - Q(s_t, a_t) \right)}_{\text{old value}}$$

$$\text{new value (temporal difference target)}$$

where r_t is the reward received when moving from the state s_t to the state s_{t+1}, and **α** is the learning rate 0< **α** <=1.

Program:

Github Links

Python Model & API

React Frontend

Data Structures and Classes

We have used python Dictionary to store the state <-> reward pair for each move that the menace takes. The Menace machine builds the dictionary with calculated rewards for each different move from the previous state, the state is arbitrarily set to 0 initially. The goal of the algorithm is to maximize the reward value in each iteration, thus, “training” itself to choose the best possible moves to maximize reward, this in turn results in choosing the “optimal” strategy for the board moves.

```
class PlayerTraining:
   def __init__(self, player_identifier):
       self.player_name = player_identifier
       self.position_state = []
       self.learning_rate = 0.2
       self.discount_rate = 0.8
       self.exploratory_move = 0.3
       self.greedy_move = 0.7
       self.position_value = {}
       self.reward_list = []
```

.

```
class TicTacToe:
   def __init__(self, player_one, player_two):
       self.board = np.zeros((TOTAL_NUMBER_OF_ROWS, TOTAL_NUMBER_OF_COLUMNS))
       self.player_one = player_one
       self.player_two = player_two
       self.game_has_ended = False
       self.winner = None
       self.latest_board_state = None
       self.player_symbol = 1  # when the player takes the first move, the symbol is 1
       self.player_one_wins = 0
       self.player_two_wins = 0
       self.draws  = 0
       self.barGraphList = []
```

We have used two classes namely

PlayerTraining

This class is used for training the model and building the pickle file which has the results of the training data in binary serialized format.

The alpha(learning rate), beta(discount rate), gamma(exploratory move), omega(greedy move) reward values are also defined here.

After tinkering with different values for the above variable, we performed more than 10 different combinations and trained the model over 35k times in each combination. We arrived at the most optimal values for the learning and exploratory rates for the reward function. The final optimized model is saved in a pickle file(binary stream), which is then used upon each api request.

TicTacToe

This class describes the board state, the players and the flag to determine the end of the game and its winner, if any, or a draw.

It also has the necessary members for the graphical representation method. The training Algorithm, choose_move function that is used to train the Menace.

```
def choose_move(self, position, current_board_state, symbol):
       # make a random move
       # get the random index from the position
       # get the the particular index from the available position using position[index]
       if np.random.uniform(0,1) <= self.exploratory_move:
           id = np.random.choice(len(position))
           choosen_move = position[id]
       else:
           # choose the move that maximizes to maximize the rewards
           max_value = -999
           for p in position:
               next_board = current_board_state.copy()
               next_board[p] = symbol
               next_board_state = self.get_latest_board_values(next_board)
               value = 0 if self.position_value.get(next_board_state) is None else self.position_value.get(next_board_state)
               if value >= max_value:
                   max_value = value
                   choosen_move = p
       logging.info("Chosen move from the computer/ player one" + str(choosen_move))
       return choosen_move
```

.

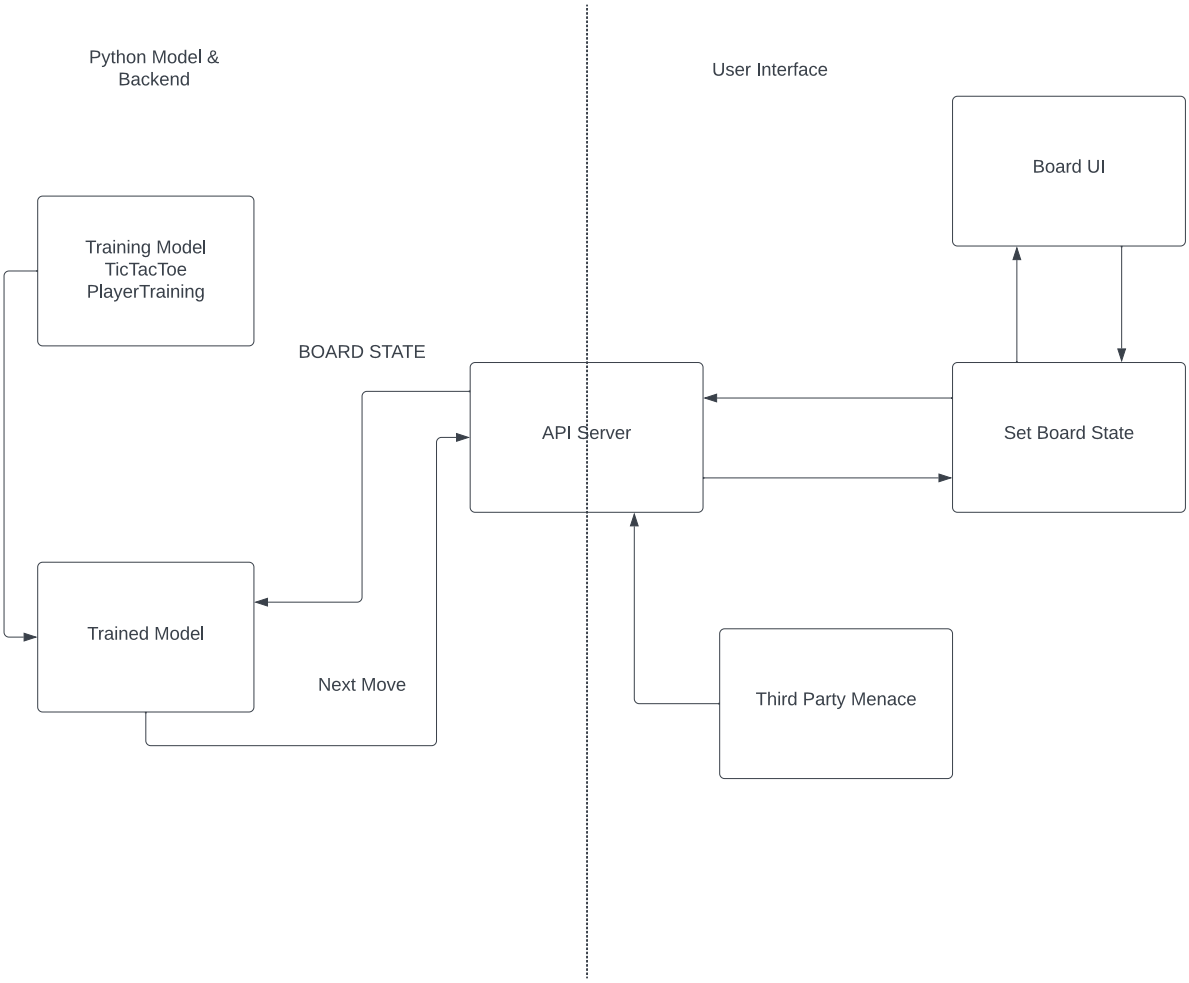
```
Algorithm pseudo code
Pseudo code

* initialize the state of the board, players and the empty q-table
* set predefined values of different parameters such as learning rate, discount rate, exploratory rate and greedy move rate
* make a random move
* calculate state value using the formula ->
          learning rate * (discount rate + reward - position_value[state])
* determine the next move by choosing the move that has the maximum reward, if none exist set the random board state value to 0
* repeat this process n number of times, with n being the number of times the model is trained
* save the states in the pickle file in binary stream
```

Invariant

The invariant in this project is a class invariant, the class TicTacToe is a class invariant because it contains the board state. The board invariant relates the number of X's and O's in the board. After each move, the number of X's or O's will change, but the relationship between them will still hold (not more O's than X's, and no more than one X more than the number of O's).

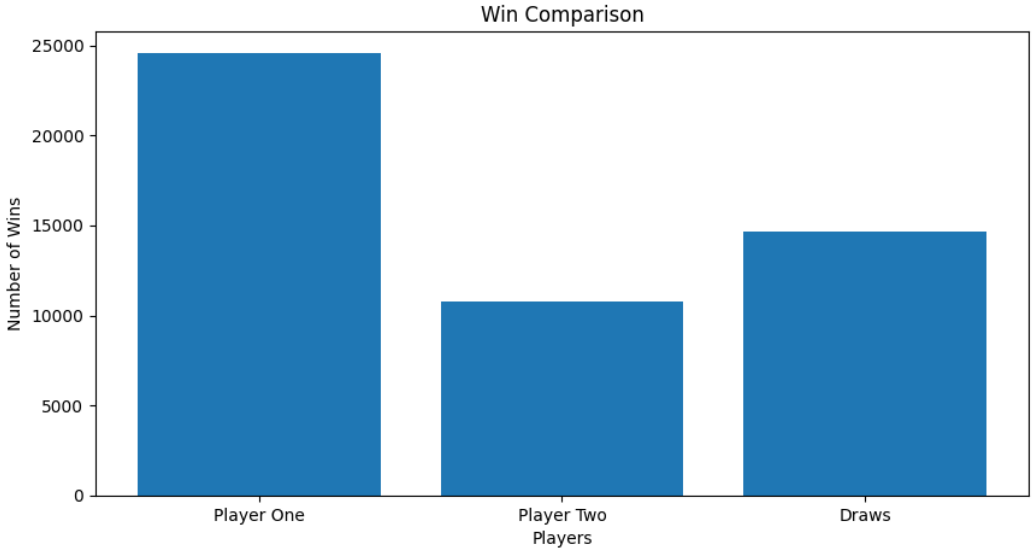
Flow Chart



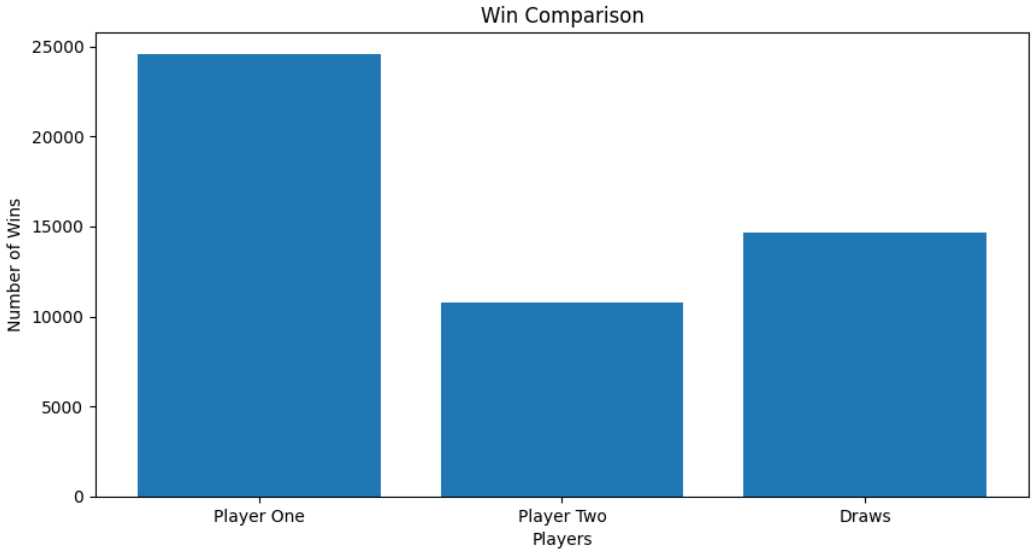
Observation And Graphical Analysis

Training the model with different reward values for the learning rate and exploratory rate for 50k runs shows different results as shown below.

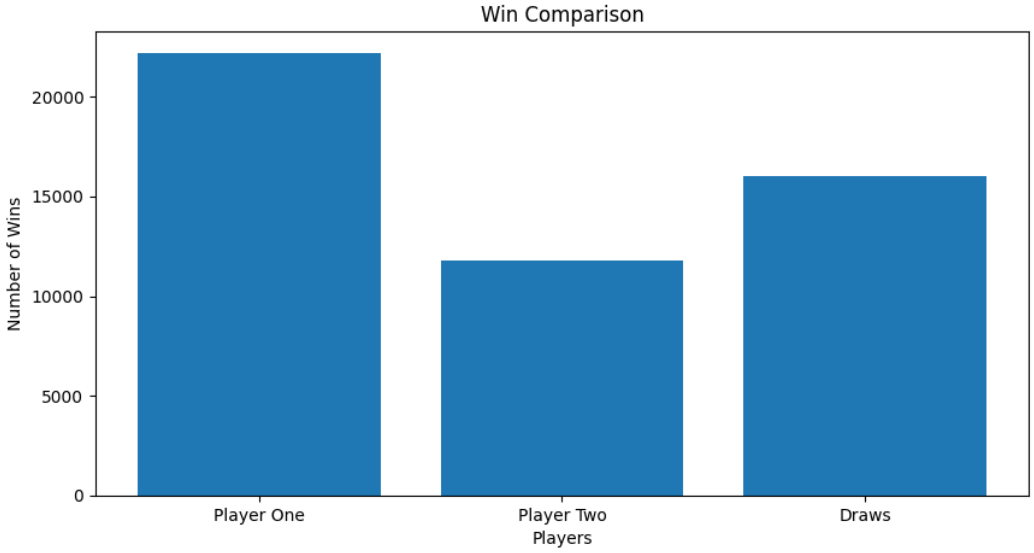
```
self.learning_rate = 0.5
self.discount_rate = 0.5
self.exploratory_move = 0.4
self.greedy_move = 0.6
```



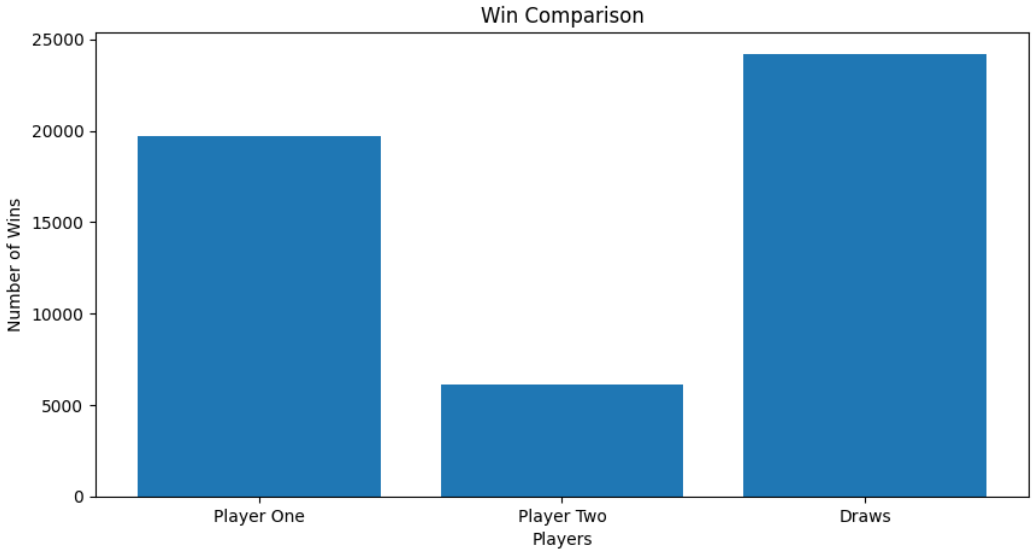

```
self.learning_rate = 0.5
self.discount_rate = 0.5
self.exploratory_move = 0.7
self.greedy_move = 0.3
```




```
self.learning_rate = 0.3
self.discount_rate = 0.7
self.exploratory_move = 0.5
self.greedy_move = 0.5
```




```
self.learning_rate = 0.4
self.discount_rate = 0.6
self.exploratory_move = 0.2
self.greedy_move = 0.8
```



Result

It can be seen that subtle variations to the learning rate and exploratory rates can have significant impact on the win percentage of player one's wins, while a non greedy approach with more reward on exploratory moves makes the menace choose random moves for each state resulting in a large number of draws. On this basis we choose the most optimum values to make the contest as even as possible by moderating the greedy approach and exploratory approach in tanden, thus unlocking the beauty of reinforcement learning.

Test Cases

Unit test cases have been added to different function in the TicTacToe class using unittest.mock library

```
# Test case for the get_latest_board_value
    def test_get_latest_board_values(self):
```

```
        self.assertEqual(tictactoe_reinforcement.PlayerTraining.get_latest_board_values(self, np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0
        self.assertEqual(tictactoe_reinforcement.PlayerTraining.get_latest_board_values(self, np.array([[1, 0, 0], [0, 0, 0], [0, 0, 0
        self.assertEqual(tictactoe_reinforcement.PlayerTraining.get_latest_board_values(self, np.array([[0, 1, 0], [0, 0, 0], [0, 0, 0
        self.assertEqual(tictactoe_reinforcement.PlayerTraining.get_latest_board_values(self, np.array([[0, 0, 1], [0, 0, 0], [0, 0, 0
        self.assertEqual(tictactoe_reinforcement.PlayerTraining.get_latest_board_values(self, np.array([[0, 0, 0], [1, 0, 0], [0, 0, 0
        self.assertEqual(tictactoe_reinforcement.PlayerTraining.get_latest_board_values(self, np.array([[0, 0, 0], [0, 1, 0], [0, 0, 0
        self.assertEqual(tictactoe_reinforcement.PlayerTraining.get_latest_board_values(self, np.array([[0, 0, 0], [0, 0, 1], [0, 0, 0
        self.assertEqual(tictactoe_reinforcement.PlayerTraining.get_latest_board_values(self, np.array([[0, 0, 0], [0, 0, 0], [1, 0, 0
        self.assertEqual(tictactoe_reinforcement.PlayerTraining.get_latest_board_values(self, np.array([[0, 0, 0], [0, 0, 0], [0, 1, 0
        self.assertEqual(tictactoe_reinforcement.PlayerTraining.get_latest_board_values(self, np.array([[0, 0, 0], [0, 0, 0], [0, 0, 1

    # Check for the available
    # Based on the board value the available positions must be calculated
    def test_available_positions(self):
        board_one =  tictactoe_reinforcement.TicTacToe("playerone", "player_two")
        board_one.board = np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]])
        assert board_one.available_position() == [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2,2)]
        board_one.board = np.array([[0, 0, 0], [0, 0, 0], [0, 0, 1]])
        assert board_one.available_position() == [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1)]
        board_one.board = np.array([[0, 0, 0], [0, 0, 0], [0, -1, 1]])
        assert board_one.available_position() == [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0)]
        board_one.board = np.array([[1, 0, 0], [0, 0, 0], [0, -1, 1]])
        assert board_one.available_position() == [(0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0)]
        board_one.board = np.array([[1, -1, 1], [0, -1, 0], [0, -1, 1]])
        assert board_one.available_position() == [(1, 0),(1, 2), (2, 0)]

    # Unit testing for checking the winners, loosers and draws
    def test_check_win(self):
        the_board = tictactoe_reinforcement.TicTacToe("playerone", "player_two")
        the_board.board = np.array([[1, 1, 1], [0, 0, 0], [0, 0, 0]])
        assert the_board.check_win() == 1
        the_board.board = np.array([[0, 0, 0], [1, 1, 1], [0, 0, 0]])
        assert the_board.check_win() == 1
        the_board.board = np.array([[0, 0, 0], [0, 0, 0], [1, 1, 1]])
        assert the_board.check_win() == 1
        the_board.board = np.array([[1, 0, 0], [1, 0, 0], [1, 0, 0]])
        assert the_board.check_win() == 1
        the_board.board = np.array([[0, 0, 1], [0, 1, 0], [1, 0, 0]])
        assert the_board.check_win() == 1
        the_board.board = np.array([[-1, -1, -1], [0, 0, 0], [0, 0, 0]])
        assert the_board.check_win() == -1
        the_board.board = np.array([[-1, 0, 0], [0, -1, 0], [0, 0, -1]])
        assert the_board.check_win() == -1
        the_board.board = np.array([[-1, 0, 0], [-1, 0, 0], [-1, 0, 0]])
        assert the_board.check_win() == -1
        the_board.board = np.array([[1, -1, 1], [-1, 1, -1], [-1,1, -1]])
        assert the_board.check_win() == 0

    # Unit testing using python mock functions
    def test_board_reset(self):
        the_board = tictactoe_reinforcement.TicTacToe("player_one", "player_two")
        the_board.board_reset()
        assert the_board.game_has_ended == False
        assert the_board.player_symbol == 1
        assert the_board.latest_board_state == None

    # Reset the board
    def test_reset_state(self):
        the_board = tictactoe_reinforcement.PlayerTraining("player_one")
        the_board.reset_state()
        assert the_board.position_state == []

    # player symbol update
    def test_update_board_state(self):
        the_board = tictactoe_reinforcement.TicTacToe("player_one", "player_two")
        the_board.player_symbol = 1
        the_board.update_board_state(1)
        assert the_board.player_symbol == -1
        the_board.player_symbol = -1
        the_board.update_board_state(1)
        assert the_board.player_symbol == 1
```

Conclusion

Tic Tac Toe when looked at mathematically has 32 (19,683) different ways the 3x3 grid can be filled in. Training the model randomly for more than 50,000 times with the specified learning rate, we are able to go through all of the “Optimal” moves that the menace has to take at any given state in the game. Once this hashmap of state response pairs is built it becomes a trivial problem for the menace machine to make the next move.

As for further improvements in the process of building the hashmap without using a reinforcement learning algorithm, we can look at specific boards under rotation and reflection to predict the next moves based on a limited amount of previous state data. More on this approach can be found here.

References

- http://incompleteideas.net/book/the-book-2nd.html - Richard S Sutton & Andrew G Barto
- https://en.wikipedia.org/wiki/Q-learning - Wikipedia
- https://en.wikipedia.org/wiki/Reinforcement_learning - Wikipedia
- https://coderbook.com/@marcus/how-to-unit-test-functions-without-return-statements-in-python/ - coderbook.com
- https://www.crows.org/blogpost/1685693/357431/The-Mathematics-of-Tic-Tac-Toe - Crows
- https://www.mscroggs.co.uk/blog/19 - mscroggs